MATLAB Tutorials

For

Mechatronics

Dr. Kevin Craig
Associate Professor of Mechanical Engineering
Department of Mechanical Engineering,
Aeronautical Engineering, and Mechanics
Rensselaer Polytechnic Institute
Troy, New York 12180
Phone: (518) 276-6671   E-mail: craigk@rpi.edu

(Written for: Matlab version 5.3, Simulink version 3.0, Control System Toolbox version 4.2)

July 21, 2000

ACKNOWLEDGEMENTS

Table of Contents

1. Before you begin

Prior to starting the tutorials contained in this booklet, take a little time to read through this introductory material.

The author would like to acknowledge that the format and some of the examples in this tutorial are based upon those contained in the MATLAB Manual from the Mathworks. The tutorials from the Mathworks are very well done, but are too long for use in Mechatronics.

1.1 What is in this tutorial?

Five short tutorials are contained in this booklet. They are :

- Basic MATLAB Tutorial
- Plotting Tutorial
- Transfer Function Tutorial
- Simulink Tutorial
- M-File Tutorial (optional)

These tutorial sections are in the order that they were intended to be used, (i.e., each tutorial builds upon the previous ones.)

1.2 Who can use this tutorial?

This tutorial was written for students and engineers in the field of Mechatronics. However, any college-level student with a level of understanding of computers and linear algebra should be able to use sections 2 and 3 of this tutorial. The last three sections of the tutorial are directed toward control system applications in MATLAB, and an understanding of the subject matter is assumed.

1.3 Why use MATLAB?

Many students will find that MATLAB is a very powerful numerical analysis tool. It can be used to evaluate complex functions, simulate dynamic systems, solve equations, and in many other applications. And now MATLAB version 5.3 can perform symbolic analysis with Symbolic toolbox (e.g., Mathematica, MathCad, Maple).

1.4 How the tutorial is structured?

This tutorial has, as much as possible, a consistent structure. Each section is intended to be an interactive tutorial. The reader should go through the tutorial while sitting at a computer terminal.

- Instructions and explanations are in this font.
- `MATLAB Commands and the expected responses are in this font and are indented.`

1.5 What is MATLAB?

MATLAB is a sophisticated mathematics and simulation environment that can be used to model and analyze dynamic systems. It handles continuous, discrete, linear, or nonlinear systems, and has extensive features for matrix manipulations. MATLAB is an open environment for which many specialized toolboxes have been developed:

- Control System toolbox
- Optimization toolbox
- System Identification toolbox
- Neural Network toolbox
- Signal Processing toolbox
- Robust Control toolbox
- Fuzzy Logic Toolbox
- and others

These toolboxes are designed to provide the user with a powerful set of analysis tools in each of their specific application areas. The success of the Control System toolbox has led to the development of Simulink. Simulink is graphical environment for modeling and simulating block diagrams and general nonlinear systems.

2. Basic MATLAB Tutorial
2.1 How MATLAB is structured

The workspace

The workspace is where all of the user's variables are stored.

The Command window

This window is a text window that appears once MATLAB is started. All user commands are issued from this window.

User Variables

The basic entity in MATLAB is the rectangular matrix (with real or complex entries). Each matrix must have a name, and the naming rules are similar to the rules for variable names in most computing languages. The exception to this is that unlike some languages, MATLAB is *case-sensitive*. In other words, variable names are sensitive to upper case and lower case. For example, if a matrix exists in the workspace named `stuff`, this matrix cannot be referred to as `STUFF`.

M-functions and Script files

M-functions and script files are often referred to under the larger category of m-files. M-files represent an important aspect of MATLAB that the user should be aware of. The full power and flexibility of MATLAB is based on these m-files.

M-files are simply text files with a ".m" extension. These files are written in the MATLAB programming language. This language is an extension of the same commands that one uses in the workspace with the addition of some program-flow-control commands.

MATLAB Data Files

MATLAB data files are binary files used to store workspace variables for later use.

Diary Files

Diary files save a record of a user's command window session in a text file (graphs are not saved). This can be extremely useful in tracking down mistakes when a long series of commands has been issued. Diary files also make it much easier to communicate your problems to a consultant (or MATLAB expert) when you ask for assistance.

MATLAB's Order of Operations

For mathematical expressions, MATLAB uses the standard order of operations: arithmetic, relational, and logical. However, when a command is issued, variable/function names have to run through a mini-gauntlet. First, the names are checked against the variables in the workspace, then the current disk directory is searched for an m-file of the same name, and finally, the entire MATLAB search path (type `help path` for more) is run through. MATLAB always uses the first occurrence. So be careful not to make any variables with the same name as a function you plan to use!

MATLAB Command Syntax

The general syntax of MATLAB commands is the following:

```
[output1,output2,...]=command_name(input1,input2,...)
```

where the command outputs are enclosed with square brackets and inputs within parentheses. If there is only one output, brackets are optional.

Handle Graphics

This is the MATLAB graphics system. It includes high-level commands for two-dimensional and three-dimensional data visualization, image processing, animation, and presentation graphics. It also includes low-level commands that allow you to fully customize the appearance of graphics as well as to build complete Graphical User Interfaces on your MATLAB applications.

Math Library

This is a vast collection of computational algorithms ranging from elementary functions like sum, sine, cosine, and complex arithmetic, to more sophisticated functions like matrix inverse, matrix eigenvalues, Bessel functions, and fast Fourier transforms.

MATLAB API

This is a library that allows you to write C and Fortran programs that interact with MATLAB. It includes facilities for calling routines from MATLAB (dynamic linking), calling MATLAB as a computational engine, and for reading and writing MAT-files.

Simulink

Simulink, a companion program to MATLAB, is an interactive system for simulating dynamic systems. It is a graphical mouse-driven program that allows you to model a system by drawing a block diagram on the screen and manipulating it dynamically. It can work with linear, nonlinear, continuous-time, discrete-time, multivariable, and multirate systems.

Editor/Debugger

The Editor/Debugger provides basic text editing operations as well as access to M-file debugging tools. The Editor/Debugger offers a graphical user interface. It supports automatic indenting and syntax highlighting; for details see the General Options section under View Menu. You can also use debugging commands in the Command Window.

2.2 Fundamentals

In this section, you will be introduced to several basic aspects of entering matrices and controlling the workspace.

Entering Simple Matrices

Matrices can be entered in several ways. Try the following and observe the resulting output.

Delimiters: The brackets [] indicate the beginning and end of a matrix. Spaces or commas are used to separate elements within a row, and semicolons are used to separate rows.

```
A=[1,2,3;4 5 6;7 8 9]
```

Alternatively, the matrix can be typed in as a matrix using the return key at the end of a row. The entry is not finished until the closed bracket is supplied.

```
A=[1 2 3
4 5 6
7 8 9]
```

The resulting output should be the same for the two lines above:

```
A=
  1 2 3
  4 5 6
  7 8 9
```

Enter the following lines and observe MATLAB's response...

```
A=[1,2,3;4,5;7,8,9]
```

What happened? Why?

```
A=[2 4 6;8 10 12;14 16 18];
```

Notice that on this last example there is a semicolon on the end of the line. This should have caused MATLAB to "not respond" with any output. Don't worry, MATLAB did record the matrix

A. The semicolon at the end of a line of input has the effect of preventing MATLAB from echoing the results to the screen. This is an important feature when very large matrices are being defined.

Matrix elements can be <u>any</u> MATLAB expression. For example :

```
x=[-1.3 sqrt(3) (1+2+3)*4/5]
```
results in

```
x=
  -1.3000 1.7321 4.8000
```

To define (or refer to) individual elements of a matrix, use the variable name and parentheses, try this:

```
x(5)=abs(x(1))
```

This produces

```
x=
  -1.3000   1.7321   4.8000   0.0000   1.3000
```

Note that the vector x has been augmented to 5 elements. Now try

```
x(4)=log10(x(6))
```

What happened? Why? It is possible to construct big matrices from smaller ones. Enter the following two lines:

```
r=[1 2 3 4 5];
y=[x;r]
```

results in

```
y=
  -1.3000 1.7321 4.8000 0.0000 1.3000
   1.0000 2.0000 3.0000 4.0000 5.0000
```
Now try:

```
w=[x,r]
```

This puts x and r side-by-side in the new row vector w.

Little matrices can be extracted from big matrices using the colon delimiter. For example :

```
z=A(2:3,1:2)
```

produces

```
z=
  8  10
 14  16
```

The statement defining z can be read as follows, "set z equal to the second through third rows and first and second columns of A." Now try a second example:

```
z=y(:,2:4)
```

results in

```
z=
   1.7321 4.8000 0.0000
   2.0000 3.0000 4.0000
```

The issued command can be interpreted as "z is equal to all rows of y and the second through fourth columns." Notice that the colon can be used to specify a range of rows and columns or it can be used to specify all of the rows and columns. This notation can lead to some very exotic submatrix references and can be used on either side of the equal sign, but it is always good practice to hand check the results to see if you're getting the expected submatrices.

More on entering MATLAB expressions

The most common format for issuing commands to MATLAB is in this form

variable = expression

However, the left side of that statement can be left out, and the results of the expression are automatically placed in a variable called `ans`. For example type:

```
1900/81
```
results in

```
ans=
   23.4568
```

The question often arises what if the expression to be entered is longer than one line on the screen? The correct way to do this is to use ellipses at the end of the current line.

For example

```
s=1 - 1/2 + 1/3 -  1/4 + 1/5 - 1/6 + 1/7 ...
     - 1/8 + 1/9 - 1/10 + 1/11 - 1/12
```

produces the expected result.

Getting workspace information

To get a list of the current variables defined in the workspace type :

```
who
```

produces a list of the currently defined variables. Try typing

```
whos
```

What is the difference? By typing `what`, a current list of m-files in the search path are shown. (Just for kicks, try `why`.)

You can also edit your workspace variables using a graphical user interface tool called Workspace Browser. You can access Workspace Browser from `File|Show Workspace` menu. In addition to your workspace variables, there are also several permanent variables that MATLAB uses. The variable `eps` is an example of such a variable. This variable represents the working precision of the computer system. Type `eps` and look at the result. Other built in variables exist e.g., `pi`, `i`, and `j`.

Numbers in MATLAB

Conventional decimal notation is used by MATLAB; i.e., 3, -99, 0.0001, 1.54e-10, and 3.13E3 are all valid ways of entering numbers. In all calculations, regardless of what is displayed, MATLAB uses double precision floating point numbers.

The traditional mathematical operators are available (e.g., +, -, *, /, and ^). In addition to these, the \ operator signifies left division (explained later) and nearly all mathematical functions found on scientific calculators can be used (e.g., cos, sin, abs, log, and log10).

MATLAB will return `Inf` when a number becomes infinite. Unlike on many other systems, MATLAB does not halt when a division by zero occurs. Instead, it carries a representation of infinity through the remaining calculations. Try typing `s=1/0`, and see the answer MATLAB gives. Another special "number" that is often seen is `NaN`. This means "Not a Number." This is produced during calculations like `0/0` or `Inf/Inf`.

Complex Numbers

Complex numbers are allowed in all mathematical expressions. The format for entering them is shown in the following examples: (This will not work as expected if `i` & `j` have been redefined by the user to something other than `sqrt(-1)`.)

```
z=3+4*i
z=3+4*j
r=1; theta=pi/4; w=r*exp(i*theta)
```

To enter a complex matrix :

```
A=[1 2;3 4] + i*[5 6;7 8]
A=[1+5*i 2+6*i;3+7*i 4+8*i]
```

Both of these result in

```
A=
   1.0000+5.0000i    2.0000+6.0000i
   3.0000+7.0000i    4.0000+8.0000i
```

Note: That in the second expression for `A`, there are no spaces around the plus signs. MATLAB would take these as separate entries.

Also Note: MATLAB will allow you to redefine the quantities `i` and `j`. If you do so you can re-create "i" by entering `i=sqrt(-1)`.

Formatting Output

When displaying matrices, MATLAB will display a matrix of exact integers as such. For example, type:

```
x=[-1 0 1]
```

this yields

```
x=
  -1   0   1
```

However, if there is at least one entry that is not an integer, there are several possible display formats. Try the following

```
x=[4/3 1.234567e-8]
```

this yields

```
x=[1.3333 0.0000]
```

Notice that the second element of x appears to be 0.0000. Since the first element is so much larger than the second, the second element IS zero in the displayed precision! Type each of the format statements in the table on next page and type x with a return to see how the display types differ. You can also set the output format from the File|Preferences menu.

format short (default)
format short e
format short g
format long
format long e
format long g
format hex
format +
format bank
format rat

With the short and long formats, if the largest element in the matrix is larger than 1000 or smaller than 0.001, a common scale factor is used. For example :

With format long or short, type

```
y=1.e20*x
```

Look carefully at the result. A common scale factor of 1.0E+020 is applied to the matrix. Now change the format to one of the "e" statements, and look at x again.

The + format is used to look at large matrices. It displays a +, -, or blank if the element is positive, negative, or zero.

Using Diary files

Type help diary for a full explanation.

Basically, you issue the statement

```
diary filename.dia
```

This will create a new file if it does not already exist, or start appending to the file if it does exist. MATLAB will start recording all command window text to the file until one of the following is typed.

- diary newfile,

- diary,

- diary off,

- or you leave MATLAB.

The diary file can be reinstated using diary on.

MATLAB Search Path

MATLAB uses a *search path* to find M-files. MATLAB's M-files are organized in directories or folders on your file system. Many of these directories of M-files are provided along with MATLAB, while others are available separately as toolboxes. You can edit the MATLAB Search Path using a graphical user interface tool called Path Browser. You can open the Path Browser from File|Set Path menu. The Path Browser lets you view and modify MATLAB's search path and see all of its files.
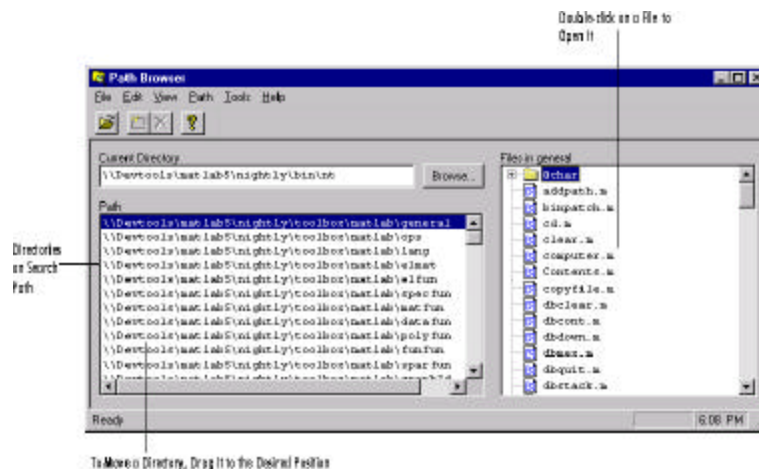
Getting HELP

To get help at any time simply type

```
help
```

If you have a specific topic in mind, e.g., how to use the plot command :

```
help plot
```

If you are really lost type

```
help help
```

In addition to `help` command there are other useful help commands in MATLAB.

- `helpwin`   Provides Windows based help
- `helpdesk` Provides comprehensive HTML help tools (search, index options). You can also reach the online PDF manuals from within heldesk.
- `demo`   Provides demo for MATLAB and related products by Math Works.
- `tour`   Provides a tour of all MATLAB products like Toolboxes, Simulink, Stateflow, Blocksets, etc.
- `lookfor`   Provides a search for keywords in M-files.

Clearing variables from the workspace

You may want to clear previously used unnecessary variables from the workspace. To clear variables and functions from memory, type `clear`.

To see its various options, type `help clear`.

It is a good practice to put this command in the beginning of m files before any other commands.

Quitting and saving the workspace

WARNING: MATLAB does not automatically save your workspace when you quit or exit!

To leave MATLAB, type `quit` or `exit`.

To save your workspace variables type

```
save
```

this saves all of your variables in a file called *MATLAB.mat*.

The full format for saving is

```
save filename variable_list
```

see `help save` for a full description.

To load an old set of workspace variables, type

```
load
```

By itself, `load` brings in the file *MATLAB.mat,* if it exists. By specifying a filename, that file will be loaded. When loading a file, the variable names used when saving are reinstated. If there is

currently a variable with the same name, it will be overwritten without warning. Any (unique) variables already in the workspace are left untouched. More can be done with the `load` command, see `help load`.

2.3 Matrix Operations

Lets perform some basic matrix arithmetic.

Transpose.

The transpose operation is signified by a single apostrophe, '. If the matrix to be transposed is complex, the result is the complex conjugate transpose. This is sometimes, but not always, the desired result for complex matrices. To get around this use `conj(A')` or `A.'`. Try the following:

```
A=[1 2 3; 4 5 6; 7 8 0]
B=A'
```

results in

```
A=
   1   2   3
   4   5   6
   7   8   0
B=
   1   4   7
   2   5   8
   3   6   0
```

and

```
x=[-1 0 2]'
```

produces

```
x=
  -1
   0
   2
```

Addition and Subtraction and Multiplication.

These operations are straightforward. As is expected, the order of multiplication and subtraction matters while it does not for addition. The matrices must be of compatible size for any operation to be carried out. The only exception to this is if one of the quantities is a scalar (a 1x1 matrix), as will be demonstrated below.

Try the following:

```
C=A+B
D=A+x
b=A*x
```

The first expression was evaluated, and the second generated an error. Why? Now try these:

```
y=x-1
w=4;z=w*y
```

Note: that the scalar quantities are applied to all elements. Now try a mix of these operations, for example :

```
x'*y
y'*x
x*y'
x*y
c=A*x+y
```

```
Make sure you understand the results of these operations.
```

Matrix Division.

Matrix division is a little more complex. This is because there is no "division" operation per se. In actuality, it involves the "best" solution of a linear system. (Scalar division rules are as expected.) There is left and right division.

It is easier to explain by example :

X=A\B is equivalent to X=inv(A)*B. It solves the problem AX=B.

X=B/A is equivalent to X=B*inv(A). It solves the problem XA=B.

If A is rectangular, A\B and B/A automatically find the least squares solution, whereas inv(A) is only valid for square matrices.

The use of matrix division is NOT recommended for simple operations because it can get confusing. Your work is easier to follow if you simply use the function inv(). For example, if you have the equation, b=A*x, and you wish to know x, type

```
x=inv(A)*b
```

Refer to the MATLAB manual tutorial for more information about matrix division.

Matrix Powers.

Raising a matrix to a matrix power, a matrix to a scalar power, a scalar to a matrix power, and a scalar to a scalar power are all possible. However, raising a matrix to a scalar power is most likely the only one you'll need. The only limitation is that the matrix must be square! Try these for practice :

```
A^4
x^3
```

Can you explain your results?

Miscellaneous Matrix Operations.

These operations are presented for your use, but their use is either too obvious or too rare to take up more space here. See the help facility or the Reference section of the MATLAB manual for more detail.

`det`	determinant of a matrix
`trace`	trace
`poly`	characteristic polynomial
`kron`	Kronecker tensor product
`expm`	exponential
`logm`	logarithm
`sqrtm`	square root
`funm`	arbitrary function

2.4 Array Operations

Array operations refer to element-by-element arithmetic operations, as opposed to matrix operations. By preceding the usual operators (e.g., / \ * ^) with a period . , the operation is carried out as an array operation. Many functions and all logical operators are considered array operations.

Addition and Subtraction.

SAME AS BEFORE.

Multiplication and Division.

For example:

```
x=[1 2 3]; y=[4 5 6];
z=x.*y
```

results in

```
z =
   4 10 18
```

Now try this :

```
z=x.\y
```

results in

```
z =
   4.0000  2.5000  2.0000
```

NOTE: The expression `x.\y` is equivalent to `y./x`

Array Powers.

Element by element powers are denoted by .^. Try these examples :

```
z=x.^y
```

produces

```
z=
   1 32 729
```

If the exponent is a scalar :

```
z=x.^2
```
produces

```
z=
   1   4   9
```

Or, the base can be a scalar :

```
z=2 .^[x y]
z=
   2   4   8   16   32   64
```

Relational and Logical Operations and Functions.

These operations (e.g., greater than, less than, ...) exist, but they are not often used outside of m-files. See the Reference section of the MATLAB manual, or help <.

Elementary Math Functions.

Many functions are inherently element by element in MATLAB. These functions are summarized below.

Trigonometric Functions		Elementary Math Functions	
sin	sine	abs	absolute value
cos	cosine	angle	phase angle
tan	tangent	sqrt	square root
asin	arcsine	real	real part
acos	arccosine	imag	imaginary part
atan	arctangent	conj	complex conjugate
atan2	four quadrant arctangent	round	round to nearest integer
sinh	hyperbolic sine	fix	round towards zero
cosh	hyperbolic cosine	floor	round towards -Inf
tanh	hyperbolic tangent	ceil	round towards +Inf
asinh	hyperbolic arcsine	sign	signum
acosh	hyperbolic arccosine	rem	remainder or modulus
atanh	hyperbolic arctangent	exp	exponential base e
		log	natural logarithm
		log10	log base 10

Generating Vectors and Matrices.

It is possible to generate vectors using operators and matrices using special functions. (Caution should be used here. If you are generating a large vector or matrix, be sure to end the statement with a semicolon. Otherwise, you'll get to see the entire vector (matrix) from beginning to end.)

To generate an evenly spaced vector try typing

```
t=1:5
```
this produces

```
t=
   1 2 3 4 5
```

Notice that this produces a row vector by default. Now try an increment other than unity:

```
y=(0:pi/4:pi)'
```
gives

```
y=
   0.0000
   0.7854
   1.5708
   2.3562
   3.1416
```

Negative increments are also possible.

Using the functions `linspace` and `logspace`, you can specify the number of points rather than the increment.

```
y=linspace(-pi,pi,4)
```

produces

```
y=
   -3.1416   -1.0472   1.0472   3.1416
```

The square identity matrix is defined by using the function `eye(n)`, where n is the number of columns or rows. The functions in the table below allow the user to generate special matrices. The arguments of these functions are either

- `(m,n)`, signifying the number of rows and columns

- `(m)`, signifying a square (m x m) matrix

or

- `(A)`, a matrix whose dimension you want to match

rand	random values
zeros	all zeros
ones	all ones

For example,

- ones(t), where t is a vector generates the unit step function.

- y(t)=3+t, for t=0, 1 2, ..., 10 is created by entering

```
t=[0:1:10];y=3*ones(t)+t
```

One last useful matrix creation function is `diag`. Here is an example of its use:

```
x=[1 2 3 4];X=diag(x)
```

produces

```
x=
   1 0 0 0
   0 2 0 0
   0 0 3 0
   0 0 0 4
```

2.5 Polynomials

Polynomials are represented as vectors containing the polynomial coefficients in descending order. The `roots` command finds roots of polynomials. For example, the roots of the polynomial s^3+2s^2+3s+4 are found by

```
p=[1 2 3 4]; roots(p)
```

produces

```
ans =
   -1.6506
   -0.1747 + 1.5469i
   -0.1747 - 1.5469i
```

The `poly` command is used to form a polynomial from its roots. It returns the coefficients of the polynomial as a row vector:

```
p=poly([-1 2])
```

produces

```
p=
   1     -1     -2
```

A polynomial can be evaluated at a point using the `polyval` command. Its syntax is

```
ps=polyval(p,s)
```

where p is a polynomial and s is the point at which the polynomial is to be evaluated. The input s can be a vector or a matrix. In such cases, the evaluation is done element by element. For example, consider the polynomial p(s)=(s+1)(s+2):

```
p=[1 3 2]; s=[1 2;3 4]; polyval(p,s)
```

gives

```
ans =
        6      12
       20      30
```

Polynomials are multiplied and divided using the `conv` and `deconv` commands, respectively. The residue command performs partial fraction expansion.

2.6 Conclusion

You should now have a basic understanding of how to use MATLAB to perform simple matrix operations. At this time, it would probably be a good idea for you to try a complete problem on your own. It is recommended that you select a problem that you know the answer to, so you can verify that you have solved it correctly. Feel free to choose one of the problems below.

2.7 Sample Problems

1. If

$$A = \begin{pmatrix} 3 & 1 & 4 \\ -2 & 0 & 1 \\ 1 & 2 & 2 \end{pmatrix} \text{ and } B = \begin{pmatrix} 1 & 0 & 2 \\ -3 & 1 & 1 \\ 2 & -4 & 1 \end{pmatrix}$$

compute:

(a) 2A   (b) A+B   (c) 2A-3B

(d) $(2A)^T$-$(3B)^T$ (e) AB   (f) BA

(g) $A^T B^T$ (h) $(BA)^T$

2. Solve the following system of equations:

$$2x_1 + x_2 + x_3 = 4$$

$$x_1 - x_2 + 2x_3 = 2$$

$$3x_1 - 2x_2 - x_3 = 0$$

3. Calculate the eigenvalues and eigenvectors for A and B in #1. (Hint: `eig`)

Answers:

1. (a) $\begin{pmatrix} 6 & 2 & 8 \\ -4 & 0 & 2 \\ 2 & 4 & 4 \end{pmatrix}$ (b) $\begin{pmatrix} 4 & 1 & 6 \\ -5 & 1 & 2 \\ 3 & -2 & 3 \end{pmatrix}$ (c) $\begin{pmatrix} 3 & 2 & 2 \\ 5 & -3 & -1 \\ -4 & 16 & 1 \end{pmatrix}$ (d) $\begin{pmatrix} 3 & 5 & -4 \\ 2 & -3 & 16 \\ 2 & -1 & 1 \end{pmatrix}$

(e) $\begin{pmatrix} 8 & -15 & 11 \\ 0 & -4 & -3 \\ -1 & -6 & 6 \end{pmatrix}$ (f) $\begin{pmatrix} 5 & 5 & 8 \\ -10 & -1 & -9 \\ 15 & 4 & 6 \end{pmatrix}$ (g) and (h) $\begin{pmatrix} 5 & -10 & 15 \\ 5 & -1 & 4 \\ 8 & -9 & 6 \end{pmatrix}$

2. $x = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^T$

3. If you used these commands

$$[\text{Avec},\text{Aval}]=\text{eig}(A)$$

$$[\text{Bvec},\text{Bval}]=\text{eig}(B)$$

you got the right answer. You should have used help eig to find out how to do this.

3. Plotting Tutorial

MATLAB has a very strong graphic capability that is well suited to scientific and engineering applications.

3.1 Creating Simple X-Y Graphs

Create the data.

First you must have some data to plot. Enter the following lines into the command window:

```
t=(0:pi/36:2*pi)';
y1=2*sin(6*t); y2=3*cos(2*t);
```

Plotting a single set of data.

To plot `y1`, enter the following command:

```
plot(y1)
```

Notice that if a vector is presented to the plot command, it plots the values against the index number of each value. The exception to this is if the vector contains complex numbers, then the vector is plotted as the imaginary part versus the real part. To illustrate this, type in the following command:

```
plot(t+2*t*i)
```

The result of this should be a straight line beginning at (0,0) and ending at (2*pi,4*pi).

To plot y1 vs. t, issue the command:

```
plot(t,y1)
```

Plotting multiple lines.

To plot multiple lines (e.g., y1 and y2), enter the following command:

```
plot([y1,y2])
```

Or to plot these same lines vs. t:

```
plot(t,[y1 y2])
```

However, for reasons that will become apparent shortly, the following command is often preferable:

```
            plot(t,y1,t,y2)
```

3.2 Taking Control of the Plots

It is OK to be able to create these plots on the screen, but usually, you need a hardcopy of the graphs as well. In this section, you will find out how to print, add titles, gridlines, and axis labels to your plots. In addition, you'll see how to control the axis scaling, proportion, and line types and colors.

Changing line types and colors.

Let's say that you wanted to plot y1 as a green line, and y2 as a series of blue circles with a dashed red line through them. You would issue the following command:

```
plot(t,y1,'g',t,y2,'bo',t,y2,'r--')
```

Notice that the format is as below, and that you cannot plot lines and symbols in the same option string.

```
plot(xdata,ydata,'options',...)
```

The full list of options is shown in the table below:

Line Types		Point Types		Colors	
–	solid	.	point	y	yellow
––	dashed	o	circle	m	magenta
:	dotted	x	x-mark	c	cyan
–.	dashdot	+	plus	r	red
		*	star	g	green
		s	square	b	blue
		d	diamond	w	white
		v	triangle (down)	k	black
		^	triangle (down)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

Try, on your own, to plot y1 as a white dotted line and y2 as a red dash-dotted line.

Adding titles, grids, and axis labels.

To add grid lines, type

```
grid
```

To add a title, type

```
title('Your title information goes here.')
```

To add axis labels, type

```
xlabel('Time (sec)')
ylabel('V (volts)')
```

If you want to add a variable's value to a string, use something like this:

```
plot(t,y1+y2,'w--')
M=200;
title(['Response @ M=' num2str(M) ' lbs.'])
```

See the MATLAB Reference guide for more about this function and using strings or type:

```
help num2str
```

Controlling the plot axis scaling.

If you noticed, MATLAB automatically selected the plot axis scales for you. This may not always produce the desired result. The command used to change this is the axis command. The axis command, like many MATLAB commands, is a multi-purpose command. For the first sample of its use, type:

```
y3=cos(t); y4=sin(t); plot(y3,y4)
```

Notice that this should be a circle, but does not look like one. Now issue the following:

```
axis('square')
```

Now it looks more like a circle. The square option of axis tells MATLAB to place the graph in a square box. Try also axis('equal'). To return to the default axis scaling with a rectangular box, type

```
axis('normal')
```

The second way to use axis is to manually specify the x-y ranges to show in the box. Type

```
axis([0 2*pi -5 5])
plot(t,y1,t,y2)
```

The format for the axis-vector is: [xmin xmax ymin ymax]

To return to the default mode with automatic ranging, type

```
Axis auto
```

Printing a graph.

The first step is to create the desired graph in the graphics window.

21

The print facility in MATLAB simply dumps the current graph window to the printer. The details on how to accomplish this vary from system to system.

To print the current contents of the Graphics window :

- From a workstation, type

```
print
```

- From Windows MATLAB

  use the menu bar in the graphics window

3.3 Advanced Graphing Stuff

Not everyone will need the following, but you'll be a better person for knowing that these options are available to you.

Clearing the graphics window, holding a plot, and adding text.

To clear the graphics window, type

```
clf
```

To hold a plot so that other lines can be added, type

```
hold on
```

And to release them so that the next plot command clears the old graph, type

```
hold off
```

To add text anywhere in the graphics window, use the `text` or `gtext` command. To find out how to use it, type `help text` or `help gtext`.

Creating logarithmic plots.

To create a log-log or semilog plot, first plot the data using the `plot` command, and then type one of the following:

```
loglog(…)
semilogx(…)
semilogy(…)
```

Making subplots.

This one is a little more complicated, but it is still not too bad. As always, refer to the manual or the help facility for more information. Let's create two separate plots, one above the other.

First, tell MATLAB that you want to divide the graphics window:

```
subplot(211)
```

This number's digits can be broken down, in order, as follows:

2 = two "rows" of plotting boxes

1 = one "column" of plotting boxes

1 = make the first plot box active, i.e., MATLAB will send the next plot command to the first box.

Windows are numbered from left to right and top to bottom.

Type

```
plot(t,y1), grid
ylabel('y1'), xlabel('t'), title('Graph 1')
subplot(212), plot(t,y2), grid
ylabel('y2'), xlabel('t'), title('Graph 2')
```

Watch the results carefully. The `subplot` command advances to the next graph box. Now type this

```
plot(t,y3)
```

Did you notice that the second graph was replaced when the third `plot` statement was issued? If you had wanted to make four separate boxes, you could have used

```
subplot(221)
```

3.4 Examples

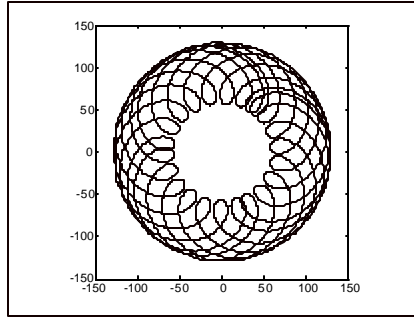(1) The equations for Mercury's orbit about the Earth are given by the following parametric equations:

$$x(t) = 93\cos t + 36\cos 4.15t$$

$$y(t) = 93\sin t + 36\sin 4.15t$$

After 7-1/3 revolutions we get the following curve called an epitrochoid. Compute the vectors x and y and plot them against each other:

```
t=[0:pi/360:2*pi*22/3];
x=93*cos(t)+36*cos(t*4.15);
y=93*sin(t)+36*sin(t*4.15);
plot(x,y),axis('square')
```
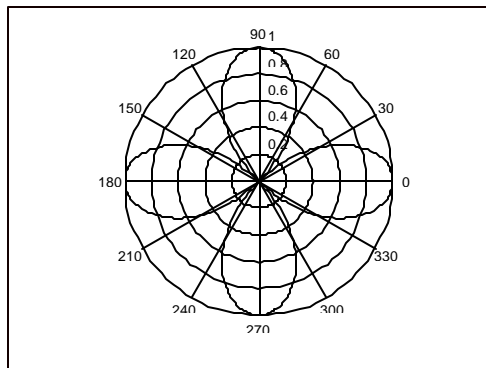
produces

(2) The equation of a four-leaf figure in polar coordinates is r=cos(2θ). The angle must be in radians for the `polar` command.
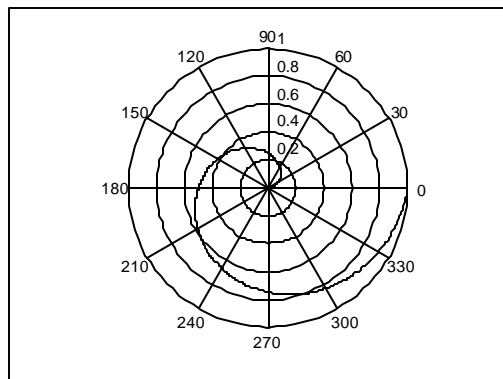
```
th=[pi/200:pi/200:2*pi]';
r=cos(2*th);
polar(th,r)
```

produces

(3) The equation for the Archimedes spiral is given by r=Kθ, K>0.

```
th=[pi/200:pi/200:2*pi]';
sa=th/(2*pi);
polar(th,sa)
```

produces

(4) The parametric equations of a circle with radius r and center at (a,b) are given by
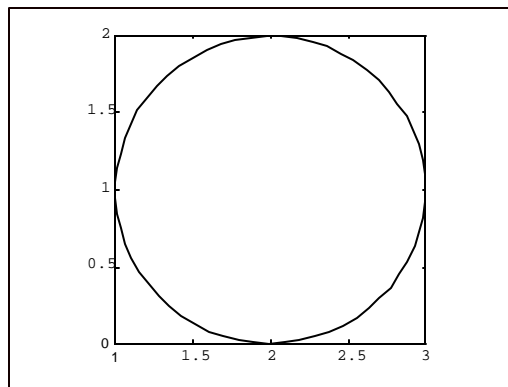
$$x(t) = r\cos(t) + a$$

$$y(t) = r\sin(t) + b$$

i.e., $$(x-a)^2 + (y-b)^2 = r^2$$

```
t=[0:0.1:2*pi];
a=2;b=1;
x=cos(t)+a;
y=sin(t)+b;
plot(x,y),axis('square')
```

produces

4. Transfer Function Tutorial

4.1 MATLAB and Transfer Functions

MATLAB, with the addition of the Control System Toolbox, can do quite a lot with respect to system analysis and simulation with transfer functions. This toolbox was written to handle the most general cases possible and can even accommodate MIMO (Multi-Input, Multi-Output) systems. This tutorial will only describe how to use MATLAB with SISO (Single-Input, Single-Output) systems. Keep this in mind when using the help facility for functions described here because some of the explanations include MIMO information.

MATLAB has very few limitations regarding transfer functions, the main one however is that the transfer function must be in polynomial form. There are, as will be discussed later, ways that MATLAB will help you get polynomial form from say pole-zero format; but nonlinear terms, like the exponential time-delay, are not valid.

In this tutorial, you will:

- learn how to manipulate TF's,

- learn how to generate Root Locus, Bode, Nyquist, and Nichols plots,

- learn how to simulate the responses of TF's,

- and be shown a complete analysis example.

This tutorial, however, makes no attempt to describe the development or "teach" these topics to the reader. It is assumed that the reader already has that knowledge.

4.2 LTI Models

You can specify linear time-invariant (LTI) systems as transfer function (TF) models, zero/pole/gain (ZPK) models, state-space (SS) models, or frequency response data (FRD) model. You can construct the corresponding models using the constructor functions.

```
sys = tf(num,den)                % transfer function
sys = zpk(z,p,k)                 % zero/pole/gain
sys = ss(a,b,c,d)                % state space
sys = frd(response, frequencies) % frequency response data
```

To find out information about LTI models, type

```
ltimodels
```

The output `sys` is a model-specific data structure called a TF, ZPK, SS, or FRD object, respectively. These objects store the model data and enable you to manipulate the LTI model as a single entity. For example, type

```
h = tf(1,[1 1])
```

This results in :

```
Transfer function:

    1
  -----
  s + 1
```

4.3 Transfer Function Basics in MATLAB

Transfer function representation

In order to handle TF's, MATLAB has a specific format that must be followed. If G(s) is your transfer function in this form (a system with m zeros and n poles) :

$$G(s) = \frac{B(s)}{A(s)} = \frac{b_0 s^m + b_1 s^{m-1} + ... + b_{m-1}s + b_m}{a_0 s^n + a_1 s^{n-1} + ... + a_{n-1}s + a_n}$$

In MATLAB, the same system would be represented by two vectors containing the coefficients of the numerator and denominator in descending powers of s:

$$B = [b_0 \quad b_1 \quad ... \quad b_{m-1} \quad b_m]$$

$$A = [a_0 \quad a_1 \quad ... \quad a_{n-1} \quad a_n]$$

For example, lets say that G(s) is to be used in MATLAB where, $G(s) = \frac{s^2 + 1}{s^3 + 2s^2 + 3s + 4}$. The numerator and denominator polynomials would be as follows, type:

```
gnum=[1 0 1]
gden=[1 2 3 4]
```

Notice that the "missing" power of s is shown as a zero in the numerator. Similarly, if you want to represent H(s)=s, hnum=[1 0] and hden=[1].

Finding poles and zeros.

Since the poles and zeros are simply the roots of the denominator and numerator, respectively, the roots command is used. For example, what are the poles and zeros of G(s) above? Type :

```
poles=roots(gden)
zeros=roots(gnum)
```

This results in :

```
poles=
  -1.6506
  -0.1747 + 1.5469i
  -0.1747 - 1.5469i
zeros=
   0 + 1.0000i
   0 - 1.0000i
```

(In case you were wondering, this command works with any polynomial, as long as the powers of the independent variable are in descending order.)

If you have defined a system as, for example, `sys = tf(gnum,gden)`, then the following commands give the same results as the above.

```
pole(sys)

zero(sys)
```

Multiplying TF's.

To multiply two polynomials together, you use the command `conv`. For example, if you have $H(s) = \dfrac{s}{s+1}$ and you want to multiply G(s) and H(s) simply issue the following commands:

```
hnum=[1 0]; hden=[1 1];
ghnum=conv(gnum,hnum)
ghden=conv(gden,hden)
```

This results in :

```
ghnum=
   1   0   1   0
ghden=
   1   3   5   7   4
```

There is a better way to "connect" two blocks in series, but this command is often useful. If, for example, you had the poles and zeros of a system, this command could be used to help create the equivalent transfer function:

```
z1=-1-i; z2=-1+i;
p1=-1; p2=-3-.5*i;  p3=-3+.5*i;
num=conv([1 -z1],[1 -z2]);
den=conv([1 -p1],conv([1 -p2],[1 -p3]));
```

results in

```
num=
   1   2   2
den=
   1.0000   7.0000   15.2500   9.2500
```

Partial fraction expansion.

To get the partial fraction expansion of a TF, you use the command `residue`. To demonstrate this, there are three examples below, each representing one of the three major cases in partial fraction expansion.

Example #1: Distinct real poles. Let $E_1(s) = \dfrac{s^3 + 5s^2 + 9s + 7}{s^2 + 3s + 2}$. To enter $E_1(s)$ into MATLAB :

```
e1num=[1 5 9 7]; e1den=[1 3 2];
```

To perform PFE:

```
[re1,pe1,ke1]=residue(e1num,e1den)
```

This results in:

```
re1=
     2
    -1
pe1=
    -1
    -2
ke1=
     1    2
```

The variable `re1` represents the numerators of the expanded fraction, the variable `pe1` represents the poles of the new denominators, and the variable `ke1` represents the constant terms. In other words, using `re1`, `pe1`, and `ke1`, the TF can be represented as follows:

$$E_1(s) = (s+2) + \frac{2}{s+1} - \frac{1}{s+2}$$

See `help residue` for a description of the general format.

Example #2: Distinct imaginary poles. Let $E_2(s) = \dfrac{s+1}{s^3 + s^2 + s}$. Enter this TF into MATLAB:

```
e2num=[1 1]; e2den=[1 1 1 0];
```
perform PFE:

```
[re2,pe2,ke2]=residue(e2num,e2den)
```

results in:

```
re2=
  -0.5000 - 0.2887i
  -0.5000 + 0.2887i
   1.0000
pe2=
  -0.5000 + 0.8660i
  -0.5000 - 0.8660i
        0
ke2=
     []
```

This corresponds to:

$$E_2(s) = \frac{-0.5 - 0.2887i}{s + 0.5 - 0.866i} + \frac{-0.5 + 0.2887i}{s + 0.5 + 0.866i} + \frac{1}{s}$$

Example #3: Repeated poles. Let $E_3(s) = \dfrac{s^2 + 4s + 6}{s^3 + 3s^2 + 3s + 1}$. Enter this TF :

```
e3num=[1 4 6]; e3den=[1 3 3 1];
```

Have MATLAB perform PFE:

```
[re3,pe3,ke3]=residue(e3num,e3den)
```

This results in:

```
re3=
    1.0000
    2.0000
    3.0000
pe3=
   -1.0000
   -1.0000
   -1.0000
ke3=
    []
```

In expanded form:

$$E_3(s) = \frac{1}{s+1} + \frac{2}{(s+1)^2} + \frac{3}{(s+1)^3}$$

This example was rigged so that you would be able to see how to create the expanded form from re3, pe3, and ke3. It is up to the user to recognize what to do with repeated poles in an answer. Be aware that the first "numerator" in a series of repeated poles corresponds to the first power denominator, and so on.

4.4 Connecting block diagrams of TF's

There are several commands that are designed explicitly for deriving system transfer functions from the connection of its component blocks. These commands are

`series`	to connect two TF's in series
`feedback`	to connect two TF's in feedback
`parallel`	to connect two TF's in parallel

To maintain brevity, this tutorial will only describe how to use these commands. The help facility for these functions is very good. Trying out examples is left to the reader. Also note, that these blocks are for SISO systems only when using transfer function notation.

For all of the following descriptions, assume that you have already defined G(s) and H(s), and that the connected system transfer function is F(s).

Series connections.

To connect G(s) and H(s) in series:

```
[Fnum,Fden]=series(gnum,gden,hnum,hden) or

Fsys=series(gsys,hsys)
```

where `gsys = tf(gnum,gden)` and `hsys = tf(hnum,hden)`

Unity feedback.

To connect G(s) in unity feedback:

```
[Fnum,Fden]=feedback(gnum,gden,1,1,sign) or

Fsys=feedback(gsys,1,1,sign)
```

where, `sign=+1` for positive feedback, and `sign=-1` for negative feedback.

General feedback.

To connect G(s) and H(s) in feedback, where G(s) is in the feedforward path, and H(s) is in the feedback path:

```
[Fnum,Fden]=feedback(gnum,gden,hnum, ...
 hden,sign) or

Fsys=feedback(gsys,hsys,sign)
```

Parallel connection.

To connect G(s) and H(s) in parallel:

```
[Fnum,Fden]=parallel(gnum,gden,hnum,hden) or

Fsys=parallel(gsys,hsys)
```

Notice that there is no `sign` option here. It is assumed that the outputs are added together.

4.5 Transfer Function Analysis Plots

Several analysis plots are available in MATLAB. The most commonly used commands are listed below. Some of these commands have useful subcommands associated with them, these are discussed in their respective sections.

rlocus	Evan's root locus plot
bode	Bode diagram
nyquist	Nyquist plot
nichols	Nichols plot

These analysis tools will be demonstrated by example.

Evan's root locus.

This function calculates (and optionally plots) the closed-loop poles of an open-loop transfer function at different gains. The gain is assumed to be applied in the feedforward path, and it is also assumed that there is negative feedback.

Assume that $G(s) = \dfrac{s+1}{s^2 + s + 1}$ in the feedforward path, and $H(s) = \dfrac{1}{s+2}$ is in the feedback path.

To make a root locus plot of the closed-loop system:

```
gsys=tf([1 1],[1 1 1]);hsys=tf(1,[1 2]);
ghsys=series(gsys,hsys);
rlocus(ghsys);
```

To find out numerical values for specific closed-loop poles, you can specify a gain vector:

```
K=0:1:100; clpols=rlocus(ghsys,K);
```

Associated commands are `rlocfind` and `sgrid`. Use help to find out more about `rlocus`, `locfind`, and `sgrid`.

Bode plots.

This command is used in a similar manner to that of `rlocus`.

Assume that we want to make an open-loop bode plot of the same G(s) and H(s) as before, type:

```
bode(ghsys)
```

Or, if we want to make a closed-loop bode plot:

```
sys=feedback(gsys,hsys,-1);
bode(sys)
```

Notice that the units of the plot are db and degrees. If numerical values for phase and gain amplitude are desired, you can make a frequency vector:

```
w=logspace(-2,2,100);
[mag,phase]=bode(sys,w);
```

Unlike the plots, the units of mag and phase are magnitude ratio and degrees. An associated command that you may want to use is `margin`. This command calculates the gain and phase margins of a system. See help on `logspace`, `margin`, and `bode`.

Nichols and Nyquist plots.

The commands for these plots are `nichols` and `nyquist`. It is no mistake that these too work in a similar manner. Try the following

```
nichols(sys)
nyquist(ghsys)
```

See also `ngrid` and `margin`.

4.6 Simulating TF System Responses

Several simulation algorithms are available to help characterize the response of a transfer function to an input. There are three main commands that you can use:

step	calculate the system's response to a step
impulse	calculate the system's response to an impulse
lsim	calculate the system's response to a user supplied input

The help facility on these commands is also very good.

Step response.

To calculate a system's unit step response and plot it, simply type :

```
step(sys)
```

To calculate a system's step response and get numerical values out, type:

```
[y,t]=step(sys);
or
t=0:.05:10; [y,x]=step(sys,t);
```

Where y is the system output, x is the state history (don't worry about it if you don't need this), and t is the time vector used during the simulation. Notice that you can specify the time vector, or let MATLAB automatically select one.

Impulse response.

To calculate a system's response to an unit impulse input and plot it, type:

```
impulse(sys)
```

To get numerical values, use the same format as for `step`.

General input response.

The only two main differences between this and the above is that 1) you must create input and time vectors, and 2) MATLAB will not automatically plot the response. Let's try a unit ramp input:

```
t=0:.05:10;
u=t;
y=lsim(sys,u,t);
or

[y,ts]=lsim(sys,u,t);
plot(ts,y,t,u);
```

This command is very powerful for simulating systems. The most difficult part becomes creating the input.

4.7 Conclusion

At this point, you should be able to use transfer functions in MATLAB reasonably well. There is a lot more that can be done with MATLAB in this manner, and you are encouraged to use MATLAB's help facility to find out more.

4.8 A Complete Example

Lets assume that we have a transfer function $G(s) = \dfrac{s+5.0}{s^3 + 4.01s^2 + 13.04s + 0.13}$. We want to create root locus plots, OL bode plots, CL bode plots, and simulate this system's CL and OL response to a step input and an impulse.

Define the transfer function:

```
num=[1 5];
den=[1 4.01 13.04 0.13];
sys=tf(num,den);
```

First find out the poles and zeros:

```
olpoles=pole(sys)
olzeros=zero(sys)
```

Second, get the root locus plot:

```
rlocus(sys)
```

Lets pick a gain to use for the closed-loop system:

```
[k,poles]=rlocfind(sys)
```

Use the mouse to pick some point along one of the branches. Notice that this function also returns the CL poles in `poles`.

Create OL bode plot:

```
bode(sys)
```

Now let's build the closed-loop system using the gain you selected with rlocfind.

```
syscl=feedback(k*sys,1,1,-1)
```

Let's take a look at the CL bode plot:

```
bode(syscl)
```

Finally lets simulate the system's CL and OL response to a step input and an impulse.

```
ysol=step(sys);
yscl=step(syscl);
yiol=impulse(sys);
yicl=impulse(syscl);
```

4.9 Rltool Example

Another useful tool for transfer function analysis and control design for single-input/single-output is the command **rltool**. Start it by typing **rltool** in the command window. The following window should appear. If you want help about a button move the mouse over the button or go to the help menu.
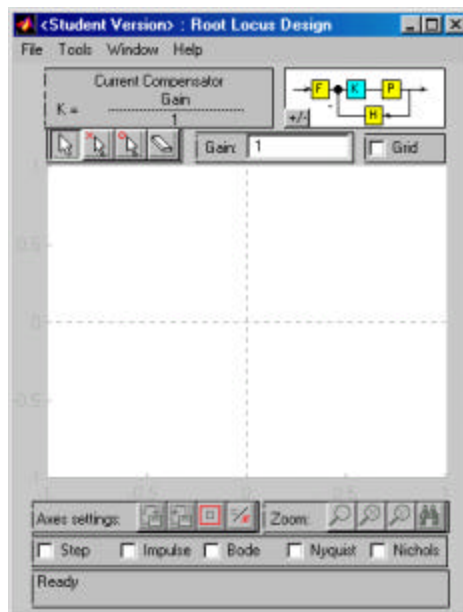
Root Locus Design Window

To use this tool import a model into the program. For this example enter the following transfer function into the command window.

$$G = \frac{0.0222}{s^2 + 0.1111s + 0.872}$$

Matlab Command
```
G = tf( 0.02222, [1 0.1111 0.872])
```
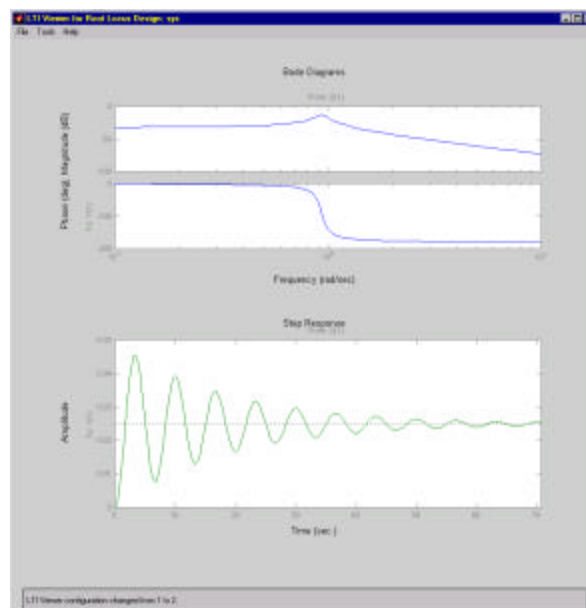
Import the model into the program,
- Select File|Import Model

The following window should appear in which you can select the model from the workspace. Import it by clicking on the arrow next to the letter P. Once you import the model you can look at the bode plot and step response by selecting the appropriate boxes.
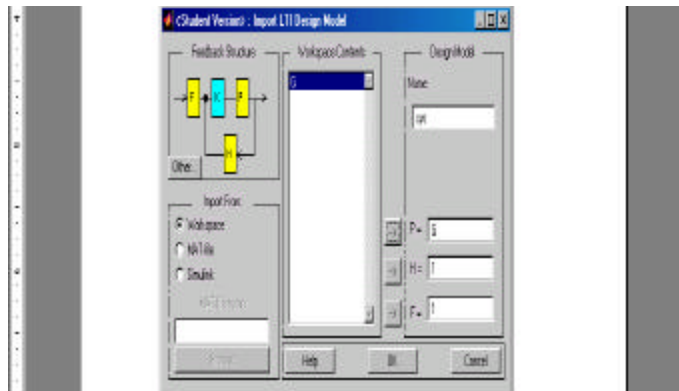

Window to Import Model

Step Response and Bode Plot

The next step is to specify the desired performance. For this example, there are two performance criteria.
- $\dot{u}_n$ greater than 1.8 for a rise time specification
- Damping ratio greater than 0.5 for an overshoot specification

Show these design specifications on the root locus
- Select Tools|Add Grid Boundary (the following figure should appear)
- Add a grid by selecting **Grid on?**
- Input the values for the natural frequency and damping ratio
- Select the appropriate boxes and click apply or OK
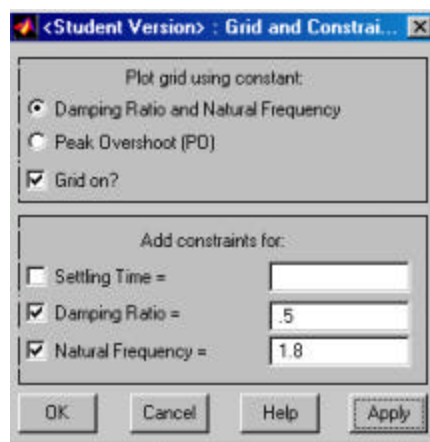

Add Design Specifications

The root locus plot should now show the acceptable region for the close loop poles. The next step is to design the compensator.
- Select Tools|Edit Compensator (The following figure should appear)
- Add a zero at –1 and a pole at –10
- Click on OK
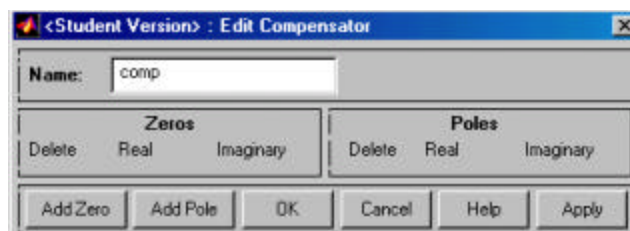
Edit Compensator Window

As you can see, the root locus changed shape, and the next step is to select an appropriate gain to move the closed loop poles. Do this by selecting the poles with the mouse and moving them to the desired location as in the following figure. You can also type the value for the gain into the corresponding box.
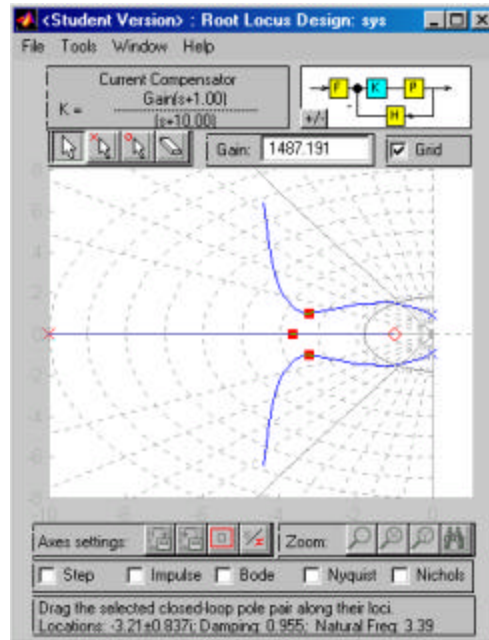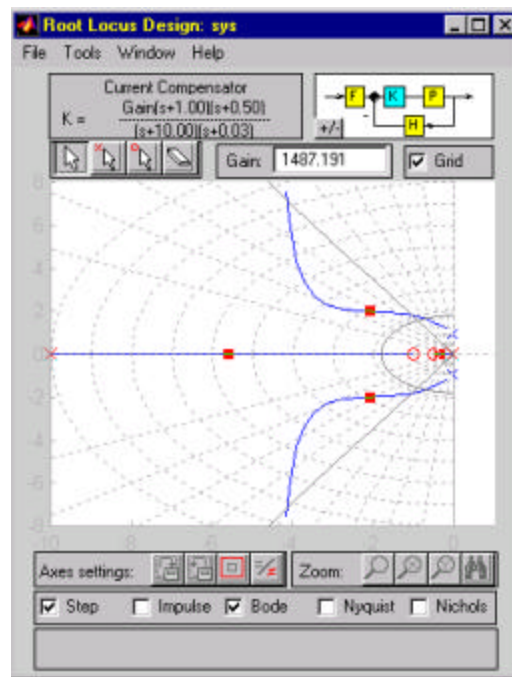

Root Locus Plot

To analyze your design, various plots are available, which are seen in the bottom of the window. Plot the step response and the bode plots for the closed loop system by selecting the appropriate boxes. They should appear like the following figure. As you can see the steady state error is too large. To improve it we will add a lag compensator.
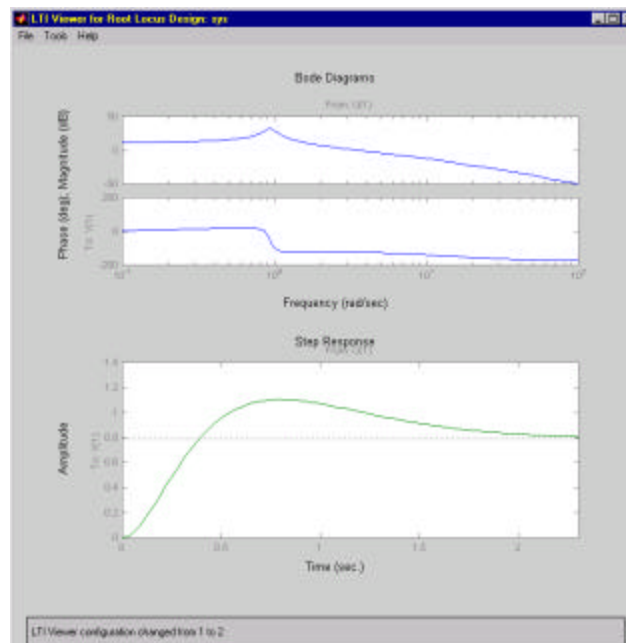
Step Response and Bode Plot

Create a lag controller by adding a pole and zero at the listed locations. The root locus should change to look like the following figure.
- zero at –0.5
- pole at –0.025

Root Locus Plot

Plot the step response and bode plot again for this system. By right clicking on the plot window a menu pops up with pull down lists. If you select **Characteristics**, more options will appear depending on the plot you selected. The following table lists the options available for the step and bode plots.

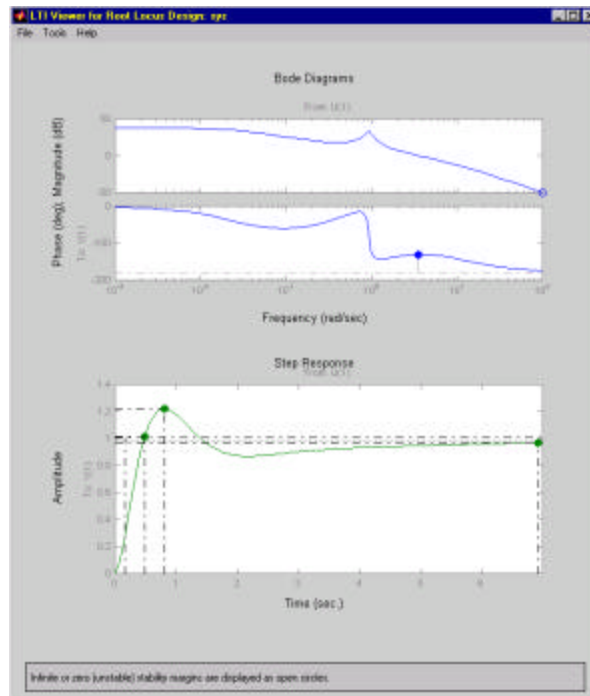| Step Response	Bode Plot
Peak Response	Peak Response
Settling Time	Stability Margins
Rise Time	
Steady State	

Display information about the rise time, overshoot, and steady state error on the step response.
- Select Characteristics|Rise Time
- Select Characteristics|Peak Response
- Select Characteristics|Settling Time

Display information about the phase margin on the bode plot.
- Select Characteristics|Stability Margins

A point appears representing the rise time, and if you click on the dot it will display the value for the rise time. You can do the same with the other points. The graphs should appear like the following figure.

Step Response and Bode Plot

Another useful feature is the ability to quickly see changes in the root locus as poles and zeros are moved. For example, select the pole at the far left and move it towards the origin. How does it change the root locus? Try this with the other poles and zeros.

5. Simulink Tutorial

5.1 What is Simulink?

Simulink is an extension of MATLAB that assists in the building of simulations. Simulink models are created much like a block diagram is created, i.e., by connecting up blocks and subsystems. Simulink is a VERY powerful simulation tool, and this tutorial is only a brief introduction to it. If you have a need for more information, refer to the Simulink manual. This tutorial is intended to

only give an introduction, and get you started as a Simulink user. With the knowledge you gain here, you should become able to use Simulink at a reasonable level of proficiency.

5.2 Simulink's Structure

To be able to use Simulink effectively, you must understand something about how the program is structured.

Blocks

A Simulink block is a subsytem with inputs and/or outputs. Within the block, some rule exists that relates the input and output. Blocks can be almost anything. Laplace transfer functions, nonlinear relations, signal sources, and signal display devices are all examples of the kinds of blocks available.

Models

A Simulink model is a collection of individual blocks, connected in such a way that they form the model for a specific system or perform a specific task.

Libraries

A Simulink library is actually a special type of model. These libraries are collections of blocks that are used by you to create other models. You can, if you like, create your own libraries of commonly used blocks simply by copying the blocks you want into a model window and saving the new "model."

Workspace

The regular MATLAB workspace is important in Simulink. It allows you to start the Simulink interface and collect simulation data for further manipulation. In the workspace, you can also control important simulation variables.

Simulation

In order to simulate a block diagram, an integration method must be chosen. The choice of method depends on the type of system to be simulated.

5.3 Simple Simulink Tasks

In this section, you will be taught the basics of Simulink. To do this, you will create a simple model, and run a simulation.

Starting Simulink.

Simulink can only be run from within MATLAB. To start Simulink, type

```
simulink
```

from the command window. You can also start Simulink by entering the name of a model that you already created.

Using the Simulink Master Library.

Once Simulink is started, a small window is opened that contains some blocks with titles under them. This is the Simulink Master Library. All of the blocks that were supplied with Simulink are accessible from this window.

Simulink's blocks are organized into categories, subcategories, subsubcategories, etc. Currently, you are looking at the top level of this library. To look at the next level down, click on the plus sign

next to the desired category, or double click the category name. All of the basic blocks are found in the *Simulink* category. For example, click on the *Simulink* plus sign, then again on the plus sign in front of *Sources*. This opens up the *source* block library. Alternatively, you can right click on the name of a category to open up the library in its own window. At some point, now or after this tutorial, you should go on a "tour" of the Simulink libraries. Look through all of the libraries to see the extent of blocks that are available. This will assist you later when you are building you own models.

To close this library, click on the minus sign next to the category name or you can select File|Close if a window is open. The same method is used to close all Simulink windows including the Master Library.

It is strongly recommended that you close the libraries once you get what you want from them. This will keep your workspace from getting cluttered, and will improve the stability of Simulink. Keeping them open gobbles up your available memory.

Now, let's create a new model for this demonstration.

- Click on the icon with a blank piece of paper

This will open a new window similar to the first, but without the blocks. Your new file is now ready for you to build your simulation.

To save a model file,

- select File|Save or File|Save As

It is recommended that you do this often. Simulink files are saved with extension, *mdl*.

Copying blocks between/within windows.

To place a block into your model, first that block must be displayed in the library. Let's copy a Signal Generator block in your file.

- Open the Sources library from the Master Library (if it is not already).
- Then, using the left mouse button, click and drag the Signal Generator block from the Sources library to your model window.
- Release it.

You have now copied this block to your file.

To copy a block within your model, you use the right mouse button.

- Click and drag the Signal Generator block using the right button somewhere else within your model.
- Release it.

This operation is a convenience when you need several copies of the same block or type of block within a model.
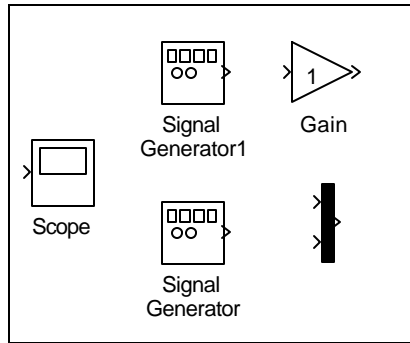
To continue the demonstration, go get copies of the following blocks:

Format: block_name (library)

- Scope (Sinks)
- Gain (Math)
- Mux (Signals and Systems)

If you have trouble finding the blocks, you can enter its name in the search box next to the binocular icon on the library browser.

Your model window would look something like this.

Deleting, moving, block parameters, connecting, branching and cleaning up model files

We do not need the second signal generator, so let's delete it.

- With a single click of the left mouse button, select the second signal generator.
- Press the delete key.

It is gone. This same method can be used to delete anything within a model, including connections and text.

Lets move the individual blocks into a more logical order (preparation for connecting them). To move a block,

- simply click and drag it to its new location with the left mouse button.

Order the blocks into approximately this configuration:
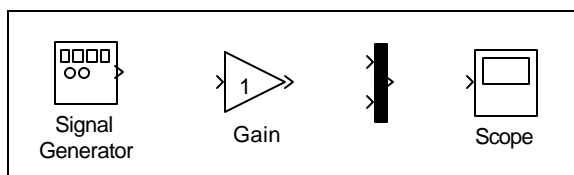
Most, but not all, blocks have block parameters associated with them. Block parameters define the behavior of the blocks, and their use varies depending on the block type. For example, let's set the gain block to have a gain of 2.5.

- Double click on the gain block to open its dialogue box.
- Edit the gain value so that it reads 2.5, and click OK.

Block parameters can also be entered as names of variables in the workspace. You can also change the number of inputs on the Mux block. Go ahead and make the change. In these dialogue boxes, you can also get help on this type of block by clicking on the help button.

To connect the blocks, you use the small arrows on the sides of the block.

- Click on the "out" arrow on the Signal Gen. block and drag the connection around. Notice that the line behaves differently in free space and near an input arrow-tail.
- Release the line in a random location. Notice that it leaves the line's end in place with an arrow head on it.
- Now click on this arrowhead and drag a new connection to the input of the gain block.

You have now connected the two blocks. (These operations also work dragging from the input side of one block to the output side of another.)

To straighten this model up, delete the connection you just made. (Select the connecting line and press delete.) Now connect up all of the blocks as shown below:
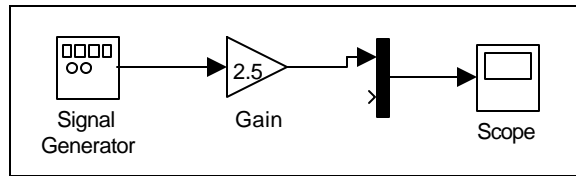
To branch (solder) a new connection line onto an existing connection, you use the right mouse button.

- Click on the **connection** from the Signal Gen. block with the right mouse button,
- drag it straight downward.
- Then using the same techniques described above, connect this branch to the second input on the Mux block as shown below.
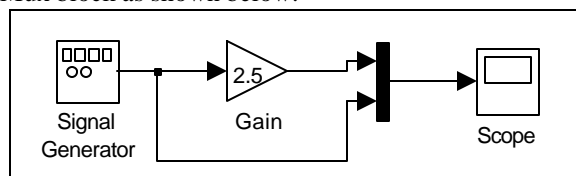
To clean up model files, it is often useful to move several blocks around and alter connecting lines. To group some blocks together for moving, copying, or deleting,

- simply click with the left mouse button and drag a rectangle around the blocks that you want to alter.
- Now the grouped features are highlighted (small handles appear in the corners of the blocks),
- and as far as editing goes, they will behave as one block. (This is different from Edit|Create Subsystem.)

To modify a connection line,

- first you select it.
- Then you can grab any of the line's vertices and move them around.

You can also change the names of the blocks to more descriptive names.

- Click on the text below the gain block.
- Now that it is highlighted, it can be edited. If you press return, it adds a new line to the name.
- To finish editing the block name, click elsewhere in the model window.

Every block must have a name, and it must be unique.

Finally, you can change the size of blocks.

- Select the gain block.
- Drag one of the handles that appear at the corners outward from the center of the block to make it larger.

If a block is made large enough, more information about the contents of that block can often be seen.

Play with these features until you get the following.

Signal
Generator

2.5

Gain

Scope
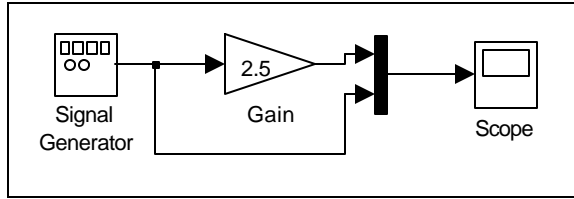
The reader is encouraged to explore Format|Flip Block and Format|Rotate Block on his/her own.

5.4 Running Simulations

The basics of running a simulation are very simple, so in addition to explaining this, you will be introduced to several other useful aspects of Simulink that are related to simulations.

Controlling a simulation

To start a simulation, select Simulation|Start from the menu bar. If you do this with the current model, nothing appears to be happening. This is because there are no 'open' display devices on the screen.

To stop the simulation, select Simulation|Stop from the menu bar.

There are several other parameters that the user can control in a simulation. These options are summarized below:

Start Time	When to start simulation (sec)
Stop Time	When to stop simulation (sec)
Solver Options	This controls the algorithm that Simulink uses to simulate your model. (see descriptions below) Variable-step solvers can modify their step sizes during the simulation. Fixed-step solvers take the same step size during the simulation.
Max Step Size	The largest time step the solver can take
Initial Step Size	A suggested first step size
Relative Tolerance	A percentage of the state's value
Absolute Tolerance	The acceptable error as the value of the measured state approaches zero

The individual solvers are described briefly below, including what they're good at solving and what they are not.

- ode45 --- The best solver to apply as a "first try" for most problems.
- ode23 --- More efficient than ode45 at crude tolerances and in the presence of mild stiffness.
- ode113 --- More efficient than ode45 at stringent tolerances
- ode15s --- Use this solver if you suspect that a problem is stiff or if ode45 failed or was very inefficient.
- ode23s --- This method can solve some kinds of stiff problems for which ode15s is not effective.
- ode23t --- Use this solver if the problem is only moderately stiff and you need a solution without numerical damping.
- ode23tb --- More efficient than ode15s at crude tolerances
- discrete (variable-step) --- The solver Simulink chooses when it detects that your model has no continuous states
- ode5 --- The fixed-step version of ode45

- ode4 --- The fourth-order Runge-Kutta formula
- ode3 --- The fixed-step version of ode23
- ode2 --- Heun's method, also known as the improved Euler formula
- ode1 --- Euler's method. Suffer from accuracy and stability problems. Not recommended for use as anything except to verify results.
- discrete (fixed-step) --- Suitable for models having no states and for which zero crossing detection and error control are not important

Watching a simulation

The scope block allows you to view a signal as it runs. To do this,

- Double click on the scope icon. This opens up a small window with a grid.
- Before we start the simulation, change the horizontal range to 10 by clicking the `Properties` icon and changing `Time range`, and the vertical range to 3 by right clicking on an axes and choosing `Properties…`. Don't close the scope unless you no longer wish to view it. Leave this one open for now.
- Change the Simulation|Parameters so that 0.01 is the maximum step size.
- Now select Simulation|Start.

You should see a sine wave on the scope's grid. Let the simulation continue while you move on to the next section.

Changing parameters "on-the-fly"

One of the nice features of Simulink is that you can alter some system parameters *while the simulation is running*. Try this.

- Double click the Signal Gen. icon.
- Select the sawtooth wave, and see how the scope graph changes.
- Try changing the amplitude of the input.

Some blocks even allow you to change their parameters while running a simulation.

- Open the gain block's dialogue box, and change the gain.

As soon as you select OK, the value has been changed. If you try to change a fixed parameter during a simulation (e.g., the order of a transfer function block), Simulink will tell you that you cannot do that or it will stop the simulation.

Play around a little, but stop the simulation before continuing on to the next section.

5.5 Using the Workspace

This section will demonstrate how to pass information back and forth between Simulink and the MATLAB workspace. To demonstrate this, we will use an entirely new example. Create a new file as shown below.
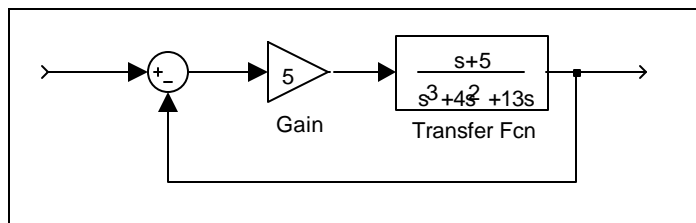
Be sure to redefine the block parameters as follows:

- Sum (Math) ... +-
- Gain (Math) ... 5
- Transfer Fcn (Continuous) ... numerator [1 5] and denominator [1 4 13 0]

With this done, you can proceed.

Sending/Receiving data between the workspace and Simulink

There are two blocks that allow you to do this:

1. to workspace (Sinks library)
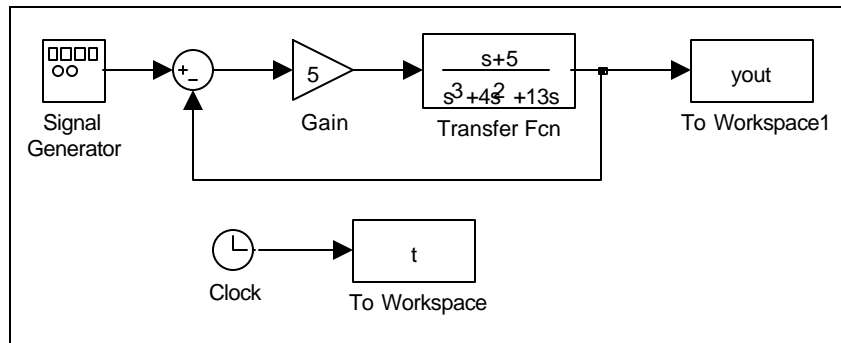
2. from workspace (Sources library)

Lets begin by using the "to workspace" block (it's a little simpler).

- Modify your model so that it appears as shown below. The clock block is found in the Sources library.
- The default variable name for To Workspace is `simout`. This is easily changed by double clicking on the block.

Be sure to change the data type from *Structure* to *Matrix*. While a *Structure* type holds more information, *Matrix* is simpler to manipulate in the workspace.

The simulation will automatically send to the workspace the time steps in the variable *tout*, but it is always a good idea to add the clock setup to any system where you use To Workspace. This is because most of the simulation algorithms use adaptive step sizing, and this is the simplest way to record the time vector associated with a simulation.

For this simulation, a maximum number of rows equal to 1000 is adequate.

Signal Generator → (+ -) → Gain: 5 → Transfer Fcn: $\dfrac{s+5}{s^3+4s^2+13s}$ → To Workspace1: yout
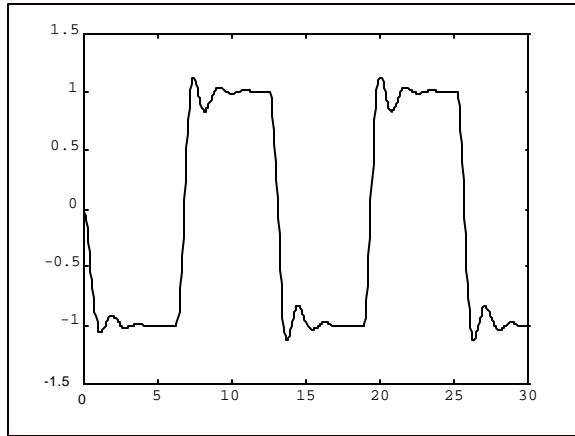
Clock → To Workspace: t

Before you start the simulation:

- Change the Signal Generator to create square waves with a frequency of 0.5 rad/sec.
- Change the Simulation|Parameters|Solver options to ode45, the stop time to 30 sec, and the max step size to 0.1 sec.
- Start the simulation.
- You'll hear a beep when the simulation is done. Now that the system output is saved in a workspace variable, it can be manipulated and plotted like any other simulation variable.

To see the results of the simulation, open the MATLAB command window and type
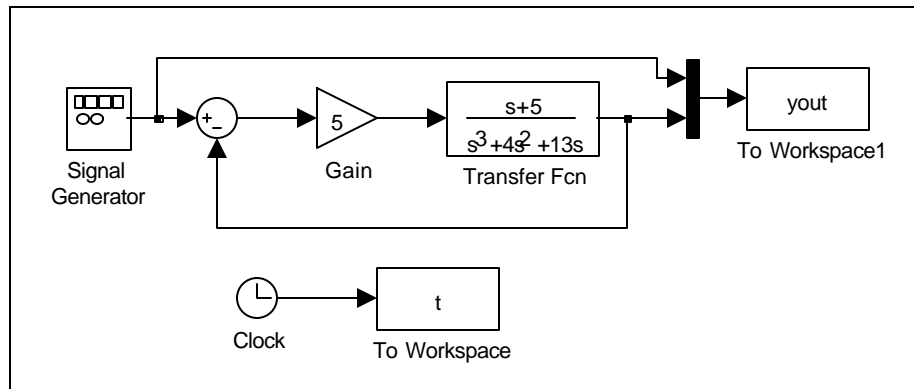
```
plot(t,yout)
```

The result looks like this:

A To Workspace block can be connected to each branch that you desire the data for (with different variable names of course). However there is a more efficient method. By using the Mux block from the Connections library, a multiple-column matrix can be stored in one variable.

For example,

- modify your model to be as shown below:

- Now run the simulation and issue the same `plot` statement in the command window.

There is now a second line on the graph. The first and second columns of `yout` correspond to the first and second inputs to the Mux block, the signal generator output and the system output.
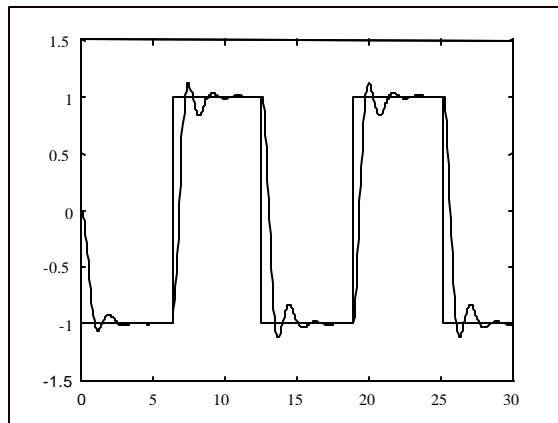
The block From Workspace works in a similar but different manner. Substitute a From Workspace block for the Signal Generator and label it as shown in the figure below. With the From Workspace block, the workspace variables, [T,U], must be defined in the workspace prior to starting the simulation.

To define [T,U], use the following statements: (don't forget the transpose operators and the ;)

```
T=[0:.05:30]';

U=(2*sin(T)+cos(3.3*T));
```

Simulink will interpolate between the data points as it simulates.

- Start the simulation.
- From MATLAB, plot the results using the same command as before. Simulink used *your* input to the system.
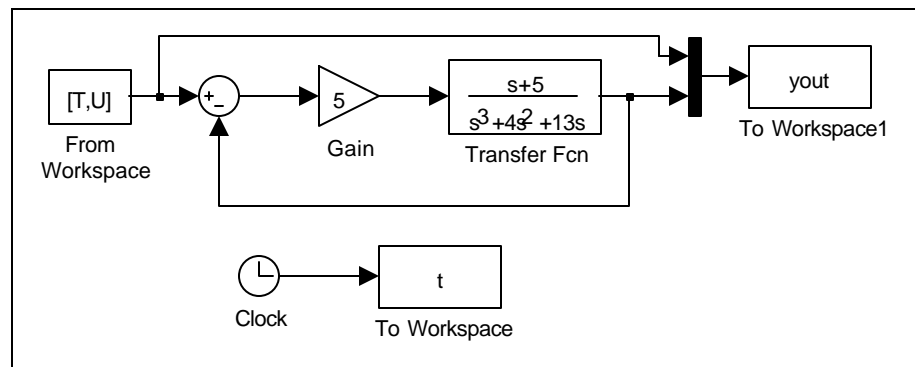
The result should be
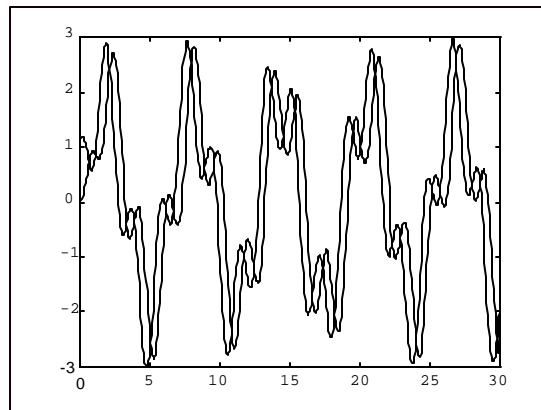
That is essentially all that you'll need to pass data back and forth between the workspace and Simulink. There are also blocks within Simulink to load values from files. It is left up to the reader to figure these out.

Getting a workspace linear model for a system.

MATLAB has a command linmod that allows the user to create a state-space model of a Simulink model file. This tutorial will only explain how to use this command with linear systems. (linmod can also be used with nonlinear systems to produce a linearized system.)

- Modify the model that you just created to the following and save it as *simtutor.mdl*.
- The In1 and Out1 blocks are contained in the *Signals and Systems* library.

1

In1

+
−

5

Gain

$$\frac{s+5}{s^3+4s^2+13s}$$

Transfer Fcn

1

Out1

Clock

t

To Workspace
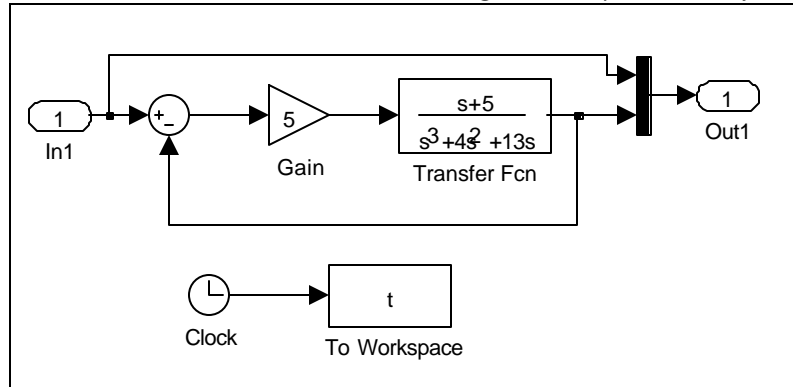
For MIMO systems, the numbers that you assign to the Inports and Outports become the order of the inputs/outputs for the `linmod` system. One limitation to keep in mind here is that Inports and Outports can only handle scalar quantities.

Now lets create the state space model for this system:

```
[A,B,C,D]=linmod('simtutor')
```

For those who wish to use transfer function notation, you can then issue the command

```
[num,den]=ss2tf(A,B,C,D)
```

See `help linmod` and `help ss2tf` for more information.

5.6 Creating Subsytems

For simulations of large systems, it is recommended that the system be broken down into smaller parts. This is done using Edit|Create subsystem from the menu bar.

- Click the left mouse button in a clear area of the window.
- Drag a rectangle from this point so that you have surrounded the entire model except the in and out ports.
- Release the button. All of the blocks except the ports will be highlighted.
- Select Edit|Create subsystem, and now the model should look like this:

You can now treat this block like any other block that you might use. You can give the block a name, copy it to other models, copy it within the same model, etc.

1

In1

1

Out1

In1   Out1
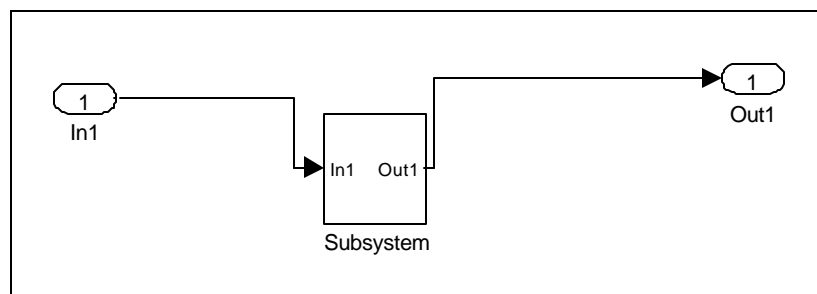
Subsystem

To see the new block's component parts and edit them,

- double click on the "grouped" block. This will open another window.

50

5.7 Masking Subsystem

By masking a subsystem you can build your own Simulink block that is composed of the built-in Simulink blocks and behaves the same way as the built-in blocks do. Masking enables you to customize the dialog box and icon for a subsystem. With masking, you can:

- Simplify the use of your model by replacing many dialog boxes in a subsystem with a single one.

- Provide a more descriptive and helpful user interface by defining a dialog box with your own block description, parameter field labels, and help text.

- Define commands that compute variables whose values depend on block parameters.

- Create a block icon that depicts the subsystem's purpose.

- Prevent unintended modification of subsystems by hiding their contents behind a customized interface.

- Create dynamic dialogs.

To create the mask for a subsystem, select the subsystem block and choose `Mask Subsystem` from the Edit menu. The *Mask Editor* pops up with which you can define dialog prompts and their characteristics, the masked block description and help text, and the commands that creates the masked block icon. For detailed instructions on the *Mask Editor*, refer to `Help` in the *Mask Editor*.

The following is an example of a masked subsystem.
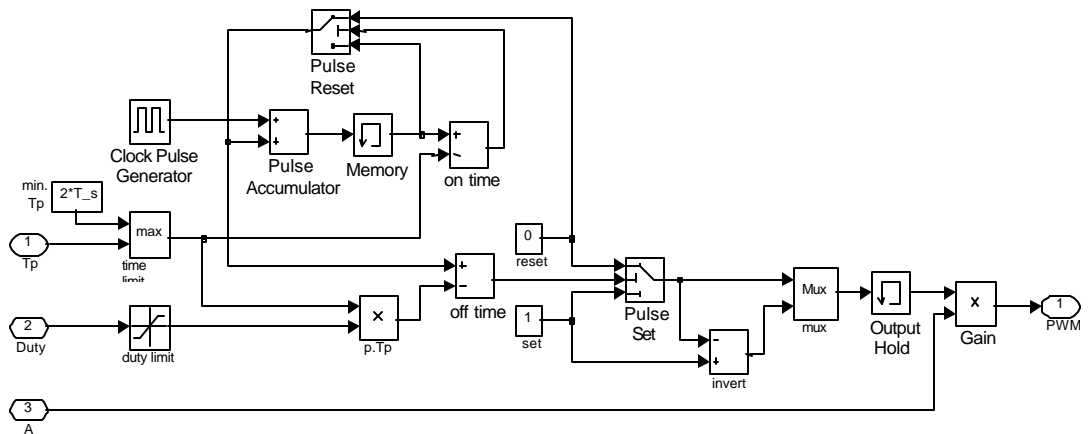
A complicated diagram made by the built-in Simulink blocks is masked into a block icon that has a dialog box of its own.
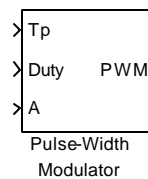
5.8 Linear Analysis Tool (LTI Viewer)

LTI Viewer is an interactive environment for comparing time and frequency responses of LTI systems. The Viewer can contain up to 6 response areas, where each response area can show a different response type and be independently manipulated. The LTI Viewer controls are found in two main locations:

1. The Figure menus (*File*, *Tools*, and *Help*)
2. Right click menus (from any axes displaying a response plot)

The Figure menus provide high level tools for manipulating data in the LTI Viewer, and configuring the appearance of the Viewer.

File allows you to Import/Export/Delete LTI Objects from the Viewer's workspace, or Open/Close/Print the Viewer and related windows.

Tools opens additional windows for configuring the number of response areas to show on the Viewer, as well as setting up response and line style preferences.

Help provides tips on using the Viewer and related windows.

Simulink provides access to Simulink model in use to get the linear model.
The right click menus provide tools for manipulating the actual responses.
The LTI Viewer can be called from Tools|Linear Analysis menu in the Simulink model file. When it is called the LTI Viewer brings *Input Point* and *Output Point* block window for use in linear analysis.

5.9 Creating Libraries

To create a library, select Library from the New submenu of the File menu. Simulink displays a new window, labeled Library: untitled. If an untitled window already appears, a sequence number is appended.
You can create a library from the command line using this command.

> new_system('newlib', 'Library')

This command creates a new library named 'newlib'. To display the library, use the open_system command. The library must be named (saved) before you can copy blocks from it.

Library (Source)   Model or Library (Destination)
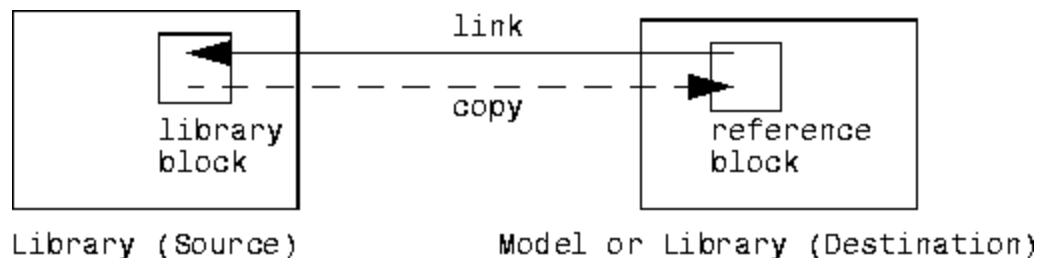
It is important to understand the terminology used with this feature.

Library - A collection of library blocks. A library must be explicitly created using `New Library` from the `File` menu.
Library block - A block in a library.
Reference block - A copy of a library block.
Link - The connection between the reference block and its library block that allows Simulink to update the reference block when the library block changes.
Copy - The operation that creates a reference block from either a library block or another reference block.

5.10 Conclusion

You should be capable of building and simulating systems using Simulink at this point. Remember that, if you have not done so, you should take some time to explore the libraries and see what is there. Practice and experience will, of course, improve your knowledge of Simulink. It is recommended that you try to model a simple system from beginning to end for practice.

There are extensive examples and demos available under the Extra library. Take a look at these to see how more complicated systems can be built.

6. *M-File Tutorial*

MATLAB supports some basic programming structures that allow looping and conditioning commands along with relational and logical operations. These new commands combined with ones already discussed can create powerful programs or new functions in MATLAB.

6.1 Introduction to m-files

An m-file is simply a text file that is created to perform a specific task. M-files can be broken down into two categories:

- Script files

- User functions

Script files are analogous to batch files. They are executed as if the user had entered the commands directly, while user functions behave like the "built in" MATLAB functions. As a matter of fact, many of the MATLAB functions are simply m-files that were written by the Mathworks.

The differences between these types of m-files are fairly simple to understand. Because script files are executed as if they were typed at the command prompt, variable references within the script file affect the workspace. Functions, on the other hand, have an input parameter list and output parameter list. Any variables used within the function are local and do not affect the workspace.

In this tutorial, you will be shown examples of both types of files and given a brief set of commands that are special to (or most commonly used in) m-files. To create your own m-files, all that you will need is an editor that creates ASCII text files. The only restriction on this editor is that it cannot insert anything other than ASCII characters into the file.

MATLAB now has its own text editor and debugger for creating and editing m-files. The editor is simple to use and has features specifically useful to MATLAB applications. You can create a new m-file by going to File|New|M-File

6.2 Script files

example1.m : A simple flow control demonstration.

This program will demonstrate the use of for-loops and if-then-elseif-else statements.

```
% This is a comment line.  The %-symbol tells MATLAB to ignore
% the rest of the line.
%
% In addition, the comments at the beginning of ANY m-file are
% considered to be the help screen for that function.  The first
% line that is not a comment terminates the help section.

% this line will not be printed if
%     help example1
% is issued from the command line.


% define some constants
t=linspace(0,1,100);
% a, b, and c are assumed to be defined in the workspace

% calculations in a for loop
for n=1:100        % for loop starts
     f(n)=t(n)^5 + a*t(n)^3 + b*t(n) + c;
end          % for loop ends

% if statement demonstration
if (any(f==0))    % are there any exact roots
     disp('An exact root for f(t) was found in [0,1].')
elseif (any(f<0) & any(f>0))  % are there any roots
     disp('f(t) has a root in [0,1]')
else  % if none of the above conditions...
     disp('f(t) has no root in [0,1]')
end

% plot results
plot(t,f), grid
```

In the workspace, enter the coefficients :

```
a=-40; b=-20; c=5;
```

Run the program by typing

```
example1
```

Change the coefficients as follows:

```
        c=100;
```

and run it again.

Finally, try the program with

```
        c=0;
```

This script file has created several variables while it was running. Type

```
        whos
```

If there had been other variables with the same name in the workspace, they would have been overwritten by the script file. Keep this in mind when you use script files.

The `disp` command displays strings and variables without displaying the variable name.

The for-loop is basically of this format:

```
        for variable=start:increment:stop

            your commands

        end
```

It is possible to nest for loops, but each one must have its own `end` statement. It is also possible to use a matrix as the loop variable. See `help for` for information on this. See `help while` for information on while-loops.

The conditional used in this program is the most general one possible. The statement must be of the following format :

```
        if (conditional)

            your commands

        elseif (conditional) (optional)

            your commands

        else (optional)

            your commands

        end
```

The `elseif` and `else` parts are optional, and you can have nested conditionals.

example2.m : A demonstration of user input/output.

```
% This file demonstrates the use of user input in
% a script file.  In addition, more complicated
% output is also demonstrated.


% get the user's desired value for search range and coeff's
tlow=input('Enter the lower bound for t : ');
tup=input('Enter the upper bound for t : ');
```

```matlab
a=input('Enter the first coeff : ');
b=input('Enter the second      : ');
c=input('Enter the third       : ');

% define some constants
t=linspace(tlow,tup,100);

% calculations in a for loop
for n=1:100        % for loop starts
     f(n)=t(n)^5 + a*t(n)^3 + b*t(n) + c;
end          % for loop ends

% define string to contain bracket e.g. '[0,5]'
T=['[' num2str(tlow) ',' num2str(tup) ']'];




% if statement demonstration
if (any(f==0))     % are there any exact roots
     disp(['An exact root for f(t) was found in ' T])
elseif (any(f<0) & any(f>0))  % are there any roots
     disp(['f(t) has a root in ' T])
else  % if none of the above conditions...
     disp(['f(t) has no root in ' T])
end

% plot results
plot(t,f), grid
```

This script file demonstrates the use of the `input` command, and shows you how to incorporate variable strings into a `disp` command. Now run the program by typing

```matlab
example2
```

and play around with the ranges and coefficients.

6.3 Functions

example3.m : A function with inputs only.

```matlab
function example3(tlow,tup,a,b,c)
% This file is a simple function with no outputs

% define some constants
t=linspace(tlow,tup,100);

% calculations in a for loop
for n=1:100        % for loop starts
     f(n)=t(n)^5 + a*t(n)^3 + b*t(n) + c;
end          % for loop ends
```

```
% define string to contain bracket e.g. '[0,5]'
T=['[' num2str(tlow) ',' num2str(tup) ']'];

% if statement demonstration
if (any(f==0))     % are there any exact roots
      disp(['An exact root for f(t) was found in ' T])
elseif (any(f<0) & any(f>0))   % are there any roots
      disp(['f(t) has a root in ' T])
else   % if none of the above conditions...
      disp(['f(t) has no root in ' T])
end

% plot results
plot(t,f), grid

return
```

This function is the same as the previous script file except that the user variables are now in a parameter list. The only way that MATLAB knows that a function is a function and not a script file is by the `function` statement at the first line in the file. This statement defines the format for the function.

To run this function type

```
example3(0,1,-40,-20,5)
```

Notice that when you run this function, the workspace variables are unaffected.

example4.m : A function with a single output.

```
function status=example4(tlow,tup,a,b,c)
% This file is a simple function with one output.

% define some constants
t=linspace(tlow,tup,100);

% calculations in a for loop
for n=1:100       % for loop starts
      f(n)=t(n)^5 + a*t(n)^3 + b*t(n) + c;
end          % for loop ends

% if statement demonstration
if (any(f==0))     % are there any exact roots
      status=0;
elseif (any(f<0) & any(f>0))   % are there any roots
      status=1;
else   % if none of the above conditions...
      status=2;
end

return
```

Notice how the `function` statement has changed. Now the function will return a number between 0 and 2 depending on whether roots were found.

Type :

```
example4(0,1,-40,-20,5)
```

You should see the following :

```
ans=
    1
```

Now type :

```
result=example4(0,1,-40,-20,0)
```

this results in

```
result=
    0
```

example5.m : A multi-output function.

```
function [root,status]=example5(tlow,tup,a,b,c)
% This file is a simple function with two outputs.

% define some constants
t=linspace(tlow,tup,100);

% calculations in a for loop
for n=1:100      % for loop starts
     f(n)=t(n)^5 + a*t(n)^3 + b*t(n) + c;
end          % for loop ends

% if statement demonstration
if (any(f==0))    % are there any exact roots
     status=0;
     root=t(find(f==0));     % return the exact roots
elseif (any(f<0) & any(f>0))  % are there any roots
     status=1;
     for n=1:99       % find brackets of roots
          if ((f(n)*f(n+1))<0)
               root=[t(n) t(n+1)];
          end
     end
else  % if none of the above conditions...
     status=2;
     root=[];
end

return
```

Notice how the function statement has changed. In this function, the roots (or bracketed ranges of the roots) and the status will be returned.

Try the following examples :

```
[r1,s1]=example5(0,1,-40,-20,0)
[r2,s2]=example5(0,1,-40,-20,5)
[r3,s3]=example5(0,1,-40,-20,100)
```

```
example5(0,1,-40,-20,5)
r4=example5(0,1,-40,-20,10)
```

Notice that if both output variables are specified, they are returned. Otherwise, only the first output variable is returned.

6.4 More M-file commands

Relational and Logical Operators			
<	less than	&	AND
<=	less than or equal	|	OR
>	greater than	~	NOT
>=	greater than or equal		
==	equal		
~=	not equal		

Flow Control Commands	
if	conditionally execute statements
elseif	used with if
else	used with if
end	terminates if, for, while statements
for	repeat statements for a number of times
while	do while
break	breaks out of for and while loops
return	return from functions
pause	pause

Programming	
`input`	get numbers from keyboard
`keyboard`	call keyboard as m-file
`error`	display error message
`function`	define function
`eval`	interpret text as command
`feval`	evaluate function given be string
`echo`	enable command echoing
`exist`	check if variable exists
`nargin`	get the number of input arguments
`nargout`	get the number of output arguments
`menu`	select item from menu
`etime`	get the elapsed time

This is only a partial list taken from the reference section of the MATLAB manual.

6.5 M-files : Some final notes

M-files can call other m-files. In fact, it is recommended that you break down a program into several routines.

Take advantage of the help section at the top of the file. It is the curse of programming that you will forget what your own functions do after a while. A well written help section will prevent many headaches for you.

If you need more instruction, the MATLAB manual has a good description of how to use m-files. In addition, you can look at the Mathworks' m-files for many of the commands. These are often helpful in finding programming tricks...

6.6 Useful M-files for Mechatronics

(1) Stepmesh.m

```
% This m-file computes the step response of a second-order system
% for values of zeta ranging from 0.1 to 1 and will create a mesh
% plot of the step responses. The natural frequency is set at 1.

n=1;
y=zeros(200,1);
i=1;
```

```
for del=0.1:0.1:1
     d=[1 2*del 1];
     t=[0:0.1:19.9]';
     y(:,i)=step(n,d,t);
     i=i+1;
end
mesh(fliplr(y), [-120 30])
```

(2) Stepchar.m

```
function[pos,tr,ts,tp]=stepchar(t,y)
% This m-file computes the % overshoot, peak time, rise time,
% and 1% settling time for a unit step response.
[mp,ind]=max(y); dimt=length(t); yss=y(dimt);
pos=100*(mp-yss)/yss; tp=t(ind);
for i=1:dimt,
     if y(i) > 1.01*yss,
           ts=t(i);
     elseif y(i) < 0.99*yss,
           ts=t(i);
     end
     end
for i=1:dimt
     if y(i) < 0.1*yss
           t1=t(i);
     elseif y(i)==mp,
           break;
     end
end
for i=1:dimt;
     if y(i) < 0.9*yss,
           t2=t(i);
     elseif y(i)==mp,
           break
     end;
end
tr=t2-t1;
```

(3) Stepmat.m

```
function [pos,tr,ts,tp]=stepmat(t,y)
% This program finds the step response characteristics
% for any number of step responses.
% The columns of y are the step responses, i.e., y=[y1 y2 ...]
[dimt,col_y]=size(y);
```

```
% This is the main loop to search over the columns of y.
     for ii=1:col_y
          yss=y(dimt,:);
% Finding the rise time, tr
     ind1=find(y(:,ii)<=0.1*yss(ii)); max(ind1); t1=t(ans);
     ind2=find(y(:,ii)<=0.9*yss(ii)); max(ind2); t2=t(ans);
     tr(ii)=t2-t1;
% Finding the 1 percent settling time, ts
     jj=dimt;
          for jj=1:dimt;
          if y(jj,ii)>=1.01*yss(ii); ts(ii)=t(jj);
          elseif y(jj,ii)<=0.99*yss(ii); ts(ii)=t(jj);
          end
     end
end
% Finding the percent overshoot, pos, and the peak time, tp.
[mp,ind]=max(y); tp=t(ind)'; pos=100*(max(y)-yss)./yss;
```

(4) Freqmat.m

```
function [mr,bw]=freqmat(w,m)
% This program calculates Mr and BW from the magnitude Bode plot.
% Use [m,w]=bode(sys); to gather the data.
% Columns of m are the absolute magnitudes, i.e., m=[m1 m2 ...]
[dimw,col_m]=size(m); mdb=ones(dimw,col_m); bw=ones(1,col_m);
for ii=1:col_m;
     jj=1;
     while m(jj,ii)>m(1,ii)/sqrt(2);

          jj=jj+1;
     end
bw(ii)=w(jj);
end
mr=20*log10(max(m));
```